Définitions algorithmiques d'outils mathématiques basiques

Programmes avec XCAS et CAML

Guillaume CONNAN

IREM de Nantes

4 janvier 2010

Sommaire

Sommaire

Sommaire

On peut définir la valeur absolue d'un réel x de la manière suivante, même si ce n'est pas l'usage en seconde :

Entrées : x(un réel)
début
  si x *positif* **alors**
    retourner x
  sinon
    retourner (-x)
fin

Algorithme 1 : valeur absolue

Sommaire

```
va(x):={
  si x>0 alors
    return(x) sinon
    return(-x)
  fsi;
}
```

Listing 1 – valeur absolue

en anglais :

```
va(x):={
  if(x>0){return(x)}
  else{return(-x)}
}
```

Listing 2 – valeur absolue (VO)

Sommaire

```
# let va(x)=
    if x>0 then x else -x;;
```

Listing 3 – valeur absolue (OCAML)

Sommaire

Sommaire

La partie entière n'est pas explicitement au programme de seconde mais sa détermination algorithmique enrichit l'étude des ensembles de nombres.
On peut définir la partie entière d'un réel x comme étant le plus grand entier inférieur à x.

Sommaire

On part du fait que la partie entière d'un nombre appartenant à $[0; 1[$ est nulle.

Ensuite, on « descend » de x vers 0 par pas de 1 si le nombre est positif en montant que $\lfloor x \rfloor = 1 + \lfloor x - 1 \rfloor$.

Si le nombre est négatif, on « monte » vers 0 en montrant que $\lfloor x \rfloor = -1 + \lfloor x + 1 \rfloor$.

On part du fait que la partie entière d'un nombre appartenant à $[0; 1[$ est nulle.
Ensuite, on « descend » de x vers 0 par pas de 1 si le nombre est positif en montant que $\lfloor x \rfloor = 1 + \lfloor x - 1 \rfloor$.
Si le nombre est négatif, on « monte » vers 0 en montrant que $\lfloor x \rfloor = -1 + \lfloor x + 1 \rfloor$.

On part du fait que la partie entière d'un nombre appartenant à $[0; 1[$ est nulle.
Ensuite, on « descend » de x vers 0 par pas de 1 si le nombre est positif en montant que $\lfloor x \rfloor = 1 + \lfloor x - 1 \rfloor$.
Si le nombre est négatif, on « monte » vers 0 en montrant que $\lfloor x \rfloor = -1 + \lfloor x + 1 \rfloor$.

Sommaire

```
partie_entiere_rec(r):={
  si(r>=0) et (r<1)
      alors 0
      sinon si r>0
            alors 1+partie_entiere_rec(r-1)
            sinon -1+partie_entiere_rec(r+1)
            fsi
      fsi
}:;
```

Listing 4 – partie entier en récursif

En anglais :

```
partie_entiere_rec(r):={
  if((r>=0) and (r<1)){0}
  else{if(r>0){1+partie_entiere_rec(r-1)}
       else{-1+partie_entiere_rec(r+1)}
      }
}:;
```

Listing 5 – partie entière en récursif (VO)

Sommaire

```
# let rec partie_entiere (r)
        =
        if r >= 0. && r < 1.
         then 0.
         else if r > 0.
                then   1. +. partie_entiere (r -. 1.)
       else -.1. +. partie_entiere (r +. 1.)
```

Listing 6 – partie entière en récursif (OCAML)

Sommaire

On peut commencer par se restreindre aux nombres positifs. On part de 0. Tant qu'on n'a pas dépassé x, on avance d'un pas. La boucle s'arrête dès que k est strictement supérieur à x. La partie entière vaut donc $k - 1$.

Entrées : x(réel positif)
Initialisation : k $\longleftarrow$ 0
début
  tant que $k<=x$ **faire**
    k$\longleftarrow$ k+1
  retourner *k-1*
fin

Algorithme 2 : partie entière d'un nombre positif

Dans le cas d'un réel négatif, il faut cette fois reculer d'un pas à chaque tour de boucle et la partie entière est la première valeur de k à être inférieure à x :

```
Entrées : x(réel)
Initialisation : k ⟵ 0
;
début
    eSi  tant que k<=x faire
     ⌊ k⟵ k+1
    retourner k-1   tant que k>x faire
     ⌊ k⟵ k-1
    retourner k
fin
```

Algorithme 3 : partie entière d'un réel quelconque

Sommaire

```
pe(x):={
local k;
k:=0;
si x>=0 alors
   tantque k<=x faire
      k:=k+1;
   ftantque;
   return(k-1);
sinon
   tantque k>x faire
      k:=k-1;
   ftantque;
   return(k);
fsi;
}:;
```

Listing 7 – partie entière en impératif

```
pe(x):={
local k;
k:=0;
if(x>=0){
   while(k<=x){k:=k+1};
   return(k-1);
        }
else{
   while(k>x){k:=k-1};
   return(k);
    }
};;
```

Listing 8 – partie entière en impératif (VO)

Sommaire

On veut arrondir un nombre réel à 10^{-2} près par défaut.

Sommaire

On va prendre la partie entière (avec l'algorithme créé précédemment) du nombre multiplié par 10^2 puis le rediviser par 10^2 :

Entrées : x(réel)
début
| **retourner** *(partie entière de* $100x$*) divisée par 100*
fin

Algorithme 4 : arrondi

Sommaire

```
arrondi(x):={
return(pe(100*x)/100.0)
}
```

Listing 9 – arrondi au centième

Sommaire

```
let arrondi_au_100_eme_par_defaut (r)
    =
    partie_entiere (100. *. r) /. 100.
```

Listing 10 – arrondi au centième avec OCAML

Sommaire

On veut arrondir à 10^{-2} près par défaut si la partie décimale est strictement inférieure à 0,5 et à 10^{-2} près par excès si la partie décimale est supérieure à 0,5.

La recherche de la partie décimale peut poser des problèmes. On commence donc par traiter le cas des nombres positifs :

Entrées : x(réel positif)
Initialisation : deci$\longleftarrow$ (x − partie entière de x)
début
  si *deci* $<$ *0,5* **alors**
    retourner *(partie entière de* $100x$*) divisée par 100*
  sinon
    retourner *0,01+(partie entière de* $100x$*) divisée par 100*
fin

Algorithme 5 : arrondi au plus proche

On peut ensuite chercher à généraliser à un réel quelconque : beaucoup d'élèves devraient se tromper ici...ce qui est formateur !

La recherche de la partie décimale peut poser des problèmes. On commence donc par traiter le cas des nombres positifs :

Entrées : x(réel positif)
Initialisation : deci$\longleftarrow$ (x − partie entière de x)
début
  si *deci* $<$ *0,5* **alors**
    retourner *(partie entière de* $100x$*) divisée par 100*
  sinon
    retourner *0,01+(partie entière de* $100x$*) divisée par 100*
fin

Algorithme 6 : arrondi au plus proche

On peut ensuite chercher à généraliser à un réel quelconque : beaucoup d'élèves devraient se tromper ici...ce qui est formateur !

La recherche de la partie décimale peut poser des problèmes. On commence donc par traiter le cas des nombres positifs :

Entrées : x(réel positif)
Initialisation : deci$\longleftarrow$ (x − partie entière de x)
début
  si *deci* $<$ *0,5* **alors**
    retourner *(partie entière de* $100x$*) divisée par 100*
  sinon
    retourner *0,01+(partie entière de* $100x$*) divisée par 100*
fin

Algorithme 7 : arrondi au plus proche

On peut ensuite chercher à généraliser à un réel quelconque : beaucoup d'élèves devraient se tromper ici...ce qui est formateur !

Sommaire

On veut par exemple calculer la somme des entiers de 1 à n.

Sommaire

Sommaire

```
som_ent(n):={
if(n==0){0}
else{n+som_ent(n-1)}
}:;
```

Listing 11 – somme des entiers en récursif

Sommaire

```
# let rec som_ent(n)=
    if n=0 then 0
    else n+som_ent(n-1);;
```

Listing 12 – somme des entiers en récursif avec OCAML

Par exemple :

```
# som_ent(100000);;
- : int = 705082704
```

On pense bien sûr à généraliser cette procédure à n'importe quelle somme du type $\sum_{k=k_0}^{n} f(k)$:

```
# let rec som_rec(f,ko,n)=
    if n=ko then f(ko)
    else f(n)+som_rec(f,ko,n-1);;
```

Listing 13 – somme des images des entiers par une fonction (CAML)

Sommaire

Entrées : n(entier naturel)
Initialisation : S$\longleftarrow$ 0// la somme est nulle au départ
début
  pour *k de 1 à n* **faire**
    S$\longleftarrow$ S+k
  retourner *S*
fin

Algorithme 8 : somme des entiers de 1 à n

Sommaire

```
som(n):={
local S;
S:=0;
pour k de 1 jusque n faire
  S:=S+k;
fpour;
return(S)
}
```

Listing 14 – somme des entiers en impératif

```
som(n):={
local S;
S:=0;
for(k:=1;k<=n;k:=k+1){S:=S+k};
return(S)
}
```

Listing 15 – somme des entiers en impératif (VO)

Sommaire

Sommaire

On peut utiliser un test :

si $x>y$;
alors
  x
sinon
  y

Algorithme 9 : maxi(x,y)

```
maxi(x,y):={
if(x>y)then{x}else{y}
}:;
```

```
# let maxi(x,y)=
    if x>y
    then x
    else y ;;
```

Un petit travail sur les valeurs absolues définies au paragraphe 3 page 4 permet de montrer que :

$$\max(x,y) = \frac{x+y+|x-y|}{2} \qquad \min(x,y) = \frac{x+y-|x-y|}{2}$$

Sommaire

Sommaire

si *liste n'a que deux éléments* **alors**
  max(les deux éléments)
sinon
  max(tête de liste,maxi(liste sans sa tête))

Algorithme 10 : maxi(liste)

```
# let rec maxilist = function
        | x::y::[] -> maxi(x,y)
        | x::queue -> maxi(x,maxilist(queue));;
```

Quand on écrit `x::y::[]`, CAML comprend qu'on a rajouté deux éléments à la liste vide : il s'agit donc d'une liste de deux éléments...

Voici une variante sans utiliser `maxi` et en créant un « garde-fou » :

```
# exception Liste_vide;;

# let rec maxbis = function
  | [] -> raise Liste_vide
  | x::[] -> x
  | x::y::queue -> if x>y then maxbis(x::queue)
                          else maxbis(y::queue);;
```

Listing 16 – maximum d'une liste en récursif (3)

Avec XCAS

```
maxlist(lx):={
 if(size(lx)==2)
   then{maxi(lx[0],lx[1])}
   else{maxi(lx[0],maxlist(tail(lx)))}
}:;
```

Sommaire

Entrées : liste
Initialisation : maxloc⟵ tête de la liste
début
  pour *k de 1 jusqu'à la taille de la liste* **faire**
    maxloc⟵ le plus grand entre maxloc et k-eme élément de la liste
fin
retourner *maxloc*

Algorithme 11 : maximum : version impérative

```
maxlist_imp(lx):={
local maxloc,k;
maxloc:=lx[0];
  for(k:=1;k<size(lx);k:=k+1){
    maxloc:=maxi(maxloc,lx[k])
  }
return(maxloc)
}:;
```

Listing 17 – maximum d'une liste en impératif